dry-rbで安全に副作用を扱おう

Press Space for next page →

dry-monadsとは？

代数的データ型をRubyで

dry-monadsは、関数型言語でおなじみの**代数的データ型**をRubyで扱えるようにするライブラリです。

提供される型

- **Result** **Success | Failure**
  - 処理の成功/失敗を表現
- **Maybe** **Some | None**
  - 値の有無を表現
- **Try** **Value | Error**
  - 例外を値として扱う

これらの型を使うことで、副作用を含む処理をいい感じに扱えるようになります。

つらいコード

```ruby
class User
  attr_reader :id, :email
  def initialize(id:, email:)
    @id = id
    @email = email
  end

  # @param id [Integer]
  # @return [User | nil] User
  def self.create(id:)
    # フィールドはnilかもしれない...
    user =
      if rand < 0.25
        new(id: "id#{id}", email: "example#{id}@example.com")
      elsif rand < 0.5
        new(id: "id#{id}", email: nil)
      ...

    # どっちもnilじゃなければインスタンスを返す
    if user.id.nil? || user.email.nil? ? nil : user
  end
end
```

インスタンス化できなかった理由を呼び出し元で知りたい

方法1. エラーにする

```ruby
class MissingIdError < StandardError; end
class MissingEmailError < StandardError; end

class User
  def self.create(id:)
    user = # 省略

    raise MissingIdError, "User ID cannot be nil" if user.id.nil?
    raise MissingEmailError, "User email cannot be nil" if user.email.nil?
    user
  end
end
```

問題点

- **ビジネスロジックで発生しうる例外はErrorとして扱うべきではない**
- **どんどん複雑になる** - 例外が増えるとコードが読みにくくなる

2. 結果をタプル(風)で返す

```ruby
class User
  def self.create(id:)
    user = # 省略

    return [:error, :missing_id] if user.id.nil?
    return [:error, :missing_email] if user.email.nil?

    [:ok, user]
  end
end

# 使用例
status, result = User.create(id: 1)
case status
when :ok
  result.save_to_db
when :error
  puts "エラー: #{result}"
end
```

問題点

- **そもそもRubyにタプルはない**

Dry-monadsを使って戻り値を包む

```ruby
class User
  extend Dry::Monads[:result]
  attr_reader :id, :email

  def initialize(id:, email:)
    @id = id
    @email = email
  end

  def self.create(id:)
    user = # 省略
    user.id.nil? || user.email.nil? ? Failure(user) : Success(user)
  end
end
```

実行してみると...

```ruby
10.times do
  User.create(id: rand(1..10)).inspect
end
```

```
Failure(#<User:0x000073d6070dce50 @id=nil, @email=nil>)
Failure(#<User:0x000073d6070dc928 @id=nil, @email="example3@example.com">)
Failure(#<User:0x000073d6070dc838 @id=nil, @email="example5@example.com">)
Failure(#<User:0x000073d6070dc748 @id=nil, @email="example9@example.com">)
Failure(#<User:0x000073d6070dc658 @id=nil, @email="example3@example.com">)
Failure(#<User:0x000073d6070dc540 @id=nil, @email="example10@example.com">)
Failure(#<User:0x000073d6070dc428 @id="id4", @email=nil>)
Failure(#<User:0x000073d6070dc2e8 @id=nil, @email="example10@example.com">)
Failure(#<User:0x000073d6070dc1d0 @id="id8", @email=nil>)
Success(#<User:0x000073d6070dc0b8 @id="id4", @email="example4@example.com">)
```

- 有効なオブジェクトなら`Success`で、無効なオブジェクトなら`Failure`で返す
- タプルのように補足情報のためにマイルールを増やす必要がない

モナドの値を取り出す

`bind`

- **Success**の場合にブロックを実行
- 成功時の処理を記述
- チェーンで `or` に繋げられる
- 新しいResultを返す必要がある

`or`

- **Failure**の場合にブロックを実行
- 失敗時の処理を記述
- `bind` の後に使用

使い方

- 成功/失敗で処理を分岐
- メソッドチェーンで記述可能
- ブロック内で値にアクセス

サンプルコード

```ruby
result = User.create(id: 1)

# bind/orパターン
result.bind do |user|
  # Successの場合
  user.save_to_db
  puts "保存しました: #{user.id}, #{user.email}"
  Success(user)
end.or do |user|
  # Failureの場合
  puts "作成失敗: #{user.inspect}"
  Failure(user)
end
```

`value_or` で仮ユーザーを作る

- `Success` -> 中身を返す
- `Failure` -> ブロックを実行して値を返す

```
result = User.create(id: 1)
user = result.value_or do |user|
  TemporaryUser.new(email: "仮@example.com")
end
user.save_to_db
```

AIとの相性について考えたこと

- 今のAIは自分で環境を作って開発ができる
- が、API連携などの外部連携はAIはひとりではどうにもならない
  - 外部連携がどういう性質を持ったものか...
  - どのようなレスポンスが「失敗」なのか...

dry-rbは情報に「文脈」を持たせる

- **Success/Failure** - 単なる値ではなく、成功か失敗かという文脈を持つ
- **エラーの詳細** - Failureに理由やコンテキストを含められる
- AIが空気を読むための情報を提供できる！

まとめ

- 副作用と向き合う協力なツール
- ActiveRecordとかでめっちゃ使いたくなる
- AI開発ととても相性が良さそう